

MULTITHREADING AND SYNCHRONIZATION

CS124 – Operating Systems
Winter 2015-2016, Lecture 10

Critical Sections

- Race conditions can be avoided by preventing multiple control paths from accessing shared state concurrently
 - Threads, processes, etc.
- A **critical section** is a piece of code that must not be executed concurrently by multiple control paths
- **Mutual exclusion**: carefully control entry into the critical section to allow only one thread of execution at a time
- Many different tools to enforce mutual exclusion in critical sections (semaphores, mutexes, read-write locks, etc.)
 - Generally, these locks block threads (passive waiting) until they can enter the critical section
- OS kernels frequently require additional tools that are compatible with use in interrupt context (i.e. nonblocking!)

Software Solutions

- A number of software solutions devised to implement mutual exclusion
- Example: Peterson's Algorithm:
 - Two processes P_0 and P_1 , repeatedly entering a critical section
 - Implementation:

```
while (true) {    // i = 0 for  $P_0$ , i = 1 for  $P_1$ ; j = 1 - i
    flag[i] = true; // State intention to enter critical section.
    turn = j;       // Let other process go first, if they want!
    while (flag[j] && turn == j);    // Wait to enter critical section.
    ...                // Critical section!
    flag[i] = false; // Leaving critical section.
    ...                // Non-critical section.
}
```

Software Solutions (2)

- Peterson's Algorithm:

- Two processes P_0 and P_1 , repeatedly entering a critical section

- Implementation:

```

while (true) {    // i = 0 for  $P_0$ , i = 1 for  $P_1$ ; j = 1 - i
    flag[i] = true; // State intention to enter critical section.
    turn = j;       // Let other process go first, if they want!
    while (flag[j] && turn == j); // Wait to enter critical section.
    ...             // Critical section!
    flag[i] = false; // Leaving critical section.
    ...             // Non-critical section.
}

```

- A process P_i can only exit the while-loop if one of these is true:

- flag[j] is false (P_j is outside the critical section)
- turn == i (it's P_i 's turn to enter the critical section)

Software Solutions (3)

- Several other software solutions to mutual exclusion:
 - Dekker's algorithm (first known correct solution)
 - Lamport's bakery algorithm
 - Syzmanski's algorithm
- All solutions are basically the same in how they work
 - Peterson's algorithm is very representative
- Problem 1: processes must busy-wait in these solutions
 - This can be changed to passive waiting without much difficulty
- Problem 2: software solutions will fail in the context of **out-of-order execution**
 - Compilers and advanced processors frequently reorder the execution of instructions to maximize pipelining/etc.

Hardware Solutions

- Modern systems provide a wide range of hardware solutions to mutual exclusion problem
- Plus, even if we want to just use the software solutions, still required to rely on specific hardware capabilities!
- Example: **barriers**
 - Previous software solutions don't work in context of out-of-order execution...
 - So, prevent out-of-order execution when it matters!
 - (Note: These are not the same as barriers in multithreading)

Optimization Barriers

- **Optimization barriers** prevent instructions before the barrier from being mixed with instructions after the barrier
 - Affects the compiler's output, not what the processor does
- Example: Linux/Pintos **barrier()** macro

```
#define barrier() asm volatile ("" : : : "memory")
```

 - Tells the compiler that the operation changes all memory locations
 - Compiler cannot rely on memory values that were cached in registers before the optimization barrier
- Can be used to ensure that one operation is actually completed before the next operation is started
 - e.g. acquiring a lock to access shared state must be completed before we can even begin interacting with the shared state
- Problem: optimization barriers do not prevent the CPU from reordering instructions at execution time...

Memory Barriers

- **Memory barriers** prevent certain kinds of instruction-reordering at the processor level
 - All instructions before the memory barrier must be completed, before any instructions after the memory barrier are started
 - (Usually, macros to impose memory barriers also impose optimization barriers, otherwise the memory barrier is useless)
- Several kinds of memory barriers, depending on the need
 - **Read memory barriers** only operate on memory-read instructions
 - **Write memory barriers** only operate on memory-write instructions
 - If unspecified, barrier affects both read and write instructions
- Also, some cases only require barriers in multiprocessor systems, not on uniprocessor systems
 - e.g. Linux has `smp_mb()` / `smp_rmb()` / `smp_wmb()` memory-barrier macros, which are no-ops on single-processor systems

Memory Barriers (2)

- Processors frequently provide multiple ways to impose memory barriers
- Example: IA32 memory-fence instructions
 - **lfence** (“load-fence”) imposes a read memory-barrier
 - **sfence** (“store-fence”) imposes a write memory-barrier
 - **mfence** (“memory-fence”) imposes a general memory-barrier
- Several other IA32 instructions also implicitly act as fences, e.g. **iret**, instructions prefixed with **lock**, etc.
- IA32 ensures that all operations before the fence are globally visible, even in multiprocessor systems
 - i.e. the system maintains cache coherency when fences are used
 - Not all architectures guarantee this...

Disabling Hardware Interrupts

- Another simple solution to preventing concurrent access is **disabling hardware interrupts**
- Frequently used to prevent interrupt handlers from manipulating shared state
 - Interrupt handlers cannot passively block, so they generally can't acquire semaphores, mutexes, etc.
 - To prevent access by an interrupt handler, just turn interrupts off
- On IA32, local interrupts are enabled and disabled via the `sti` and `cli` instructions (Set/Clear Interrupt Flag)
- Note: on multiprocessor systems, this only affects the processor that executes the instruction!
 - Other processors will continue to receive and handle interrupts

Spin Locks

- It's possible to disable interrupt handling on all processors
 - Not recommended: greatly reduces system concurrency!
- A much better approach is to use **spin locks**
- Locking procedure:
 - If the lock is immediately available, it is acquired!
 - If the lock is not immediately available, it is actively polled in a tight loop (called “spinning” on the lock) until it becomes available
- Spin locks only make sense on multiprocessor systems
 - On single-core systems they just waste CPU time, or wait forever...
- Two scenarios prompt spin-lock use:
 - Cannot context-switch away from control path (interrupt context), or
 - Lock is expected to be held for a short time, and want to avoid the overhead of a context-switch

Spin Locks and Interrupt Handlers

- Interrupt handlers can use spin locks on multiprocessor systems to guard shared state from concurrent access
 - Acquire the spin lock, access shared state, release the spin lock
- Example: a timer interrupt being triggered on each CPU
 - Each CPU executes its own timer interrupt handler...
 - Timer interrupt handler needs to access shared state!
e.g. process ready-queue, waiting queues, etc.
 - Need to enforce a critical section on manipulation of shared state
- Timer interrupt handler can acquire a spin lock...
 - Interrupts are supposed to complete quickly...
 - Even if multiple CPUs have timer interrupts occur at same time, a given handler won't wait long for handlers on other CPUs to finish

Spin Locks and Interrupt Handlers (2)

- Must be extremely careful using spin locks when control paths can be nested!
 - Scenario: multiprocessor system, nested kernel control paths
- Example: guard state that's shared between a trap handler and an interrupt handler, with a spin-lock
 - Trap handler acquires the spin lock
 - Trap handler begins accessing shared state
 - Interrupt fires! Handler attempts to acquire the same spin lock
- The system becomes deadlocked:
 - Trap handler holds a lock that interrupt handler needs to proceed
 - Interrupt handler holds CPU, which trap handler needs to proceed
 - Nobody makes any progress. ☹️
- For these situations, must also disable local interrupts before acquiring the spin lock, to avoid deadlocks

Spin Lock Guidelines

- Spin locks are only useful on multiprocessor systems
 - On single-processor systems, simply disable interrupt processing
- Spin locks should be held only for a short time
- If a critical section will only be entered by control paths running on different CPUs, simple spin locks will suffice
 - e.g. shared state only accessed from one interrupt handler, that runs on all CPUs in the system
- If more than one control path on the same CPU can enter a critical section, must disable interrupts before locking
 - e.g. shared state accessed from trap handlers + interrupt handlers
- Linux spinlock primitives:
 - `spin_lock_irq()` and `spin_unlock_irq()` disable/reenable interrupts
 - `spin_lock()` and `spin_unlock()` simply acquire/release the lock

Locks and Deadlocks

- Locking mechanisms for synchronization introduce the possibility of multiple processes entering into deadlock
 - A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause.
- Requirements for deadlock:
 - **Mutual exclusion**: resources must be held in non-shareable mode
 - **Hold and wait**: a process must be holding one resource, and waiting to acquire another resource that is currently unavailable
 - **No preemption**: a resource cannot be preempted; the process must voluntarily release the resource
 - **Circular wait**: the set of processes $\{P_1, P_2, \dots, P_n\}$ can be ordered such that P_1 is waiting for a resource held by P_2 , P_2 is waiting for a resource held by P_3 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_1

Dealing with Deadlock

- Several ways to deal with deadlock
- **Deadlock prevention:** engineer the system such that deadlock never occurs
 - Usually focuses on breaking either the “no preemption” or the “circular wait” requirement of deadlock
 - No preemption: if a process cannot acquire a resource, it relinquishes its lock on all its other resources
 - (usually not practical in practice)
 - Circular wait: impose a total ordering over all lockable resources that all processes must follow
 - As long as resources are only locked in the total ordering, deadlock can never occur
 - If a process acquires a later resource in the ordering, then wants an earlier resource in the ordering, must release all its locks and start over
 - Usually not imposed by the OS; must be imposed by the programmer

Dealing with Deadlock (2)

- **Deadlock avoidance:** the system selectively fails resource-requests in order to prevent deadlocks
 - System detects when allowing a request to block would cause a deadlock, and reports an immediate failure on the request
 - Several algorithms to do this, e.g. **Banker's algorithm**
 - Also **wound/wait** and **wait/die** algorithms:
 - Given an older process P_O and a younger process P_Y
 - Wound/wait:
 - If P_O needs a resource that P_Y holds, P_Y dies
 - If P_Y needs a resource that P_O holds, P_Y waits
 - Wait/die:
 - If P_O needs a resource that P_Y holds, P_O waits
 - If P_Y needs a resource that P_O holds, P_Y dies
- **Deadlock detection and resolution:** simply allow deadlock!
 - When a set of processes enters into deadlock, system identifies that deadlock occurred, and terminates a deadlocked process
 - (not used in operating systems; used heavily in database systems)

Semaphores

- **Semaphores** are a common synchronization mechanism
 - Devised by Edsger Dijkstra
- Allows two or more processes to coordinate their actions
- Typically, processes block until acquiring the semaphore
 - Can't use semaphores in interrupt context
- Each semaphore has this state:
 - An integer variable **value** that cannot be negative
 - A list of processes/threads waiting to acquire the semaphore
- Two operations: **wait()** and **signal()**
 - Also called **down()** and **up()**
 - Dijkstra's names:
 - P (for "prolaag," short for "probeer te verlagen" or "try to decrease")
 - V (for "verhogen" or "increase")

Semaphores (2)

- Example **wait()** implementation:
 - while `sem.value == 0`:
 - add this thread to `sem.waiting` list
 - passively block the thread
 - `sem.value := sem.value - 1`
- Example **signal()** implementation:
 - `sem.value := sem.value + 1`
 - if `sem.waiting` list is not empty:
 - `t = remove a thread from sem.waiting`
 - unlock `t`
- These operations must be enclosed in critical sections!
 - e.g. Pintos turns off interrupt handling inside **down()** and **up()**
- Blocked threads are managed in various ways
 - e.g. always put blocked threads at end of waiting, unblock from front
 - e.g. choose a random thread to unblock
 - (Sometimes, making things fair is more expensive...)

Counting Semaphores

- A semaphore's value represents how many times `wait()` can be called without blocking
 - Use it to represent e.g. how much of a given resource is available
- Called **counting semaphores** when used in this way
 - Maximum value of semaphore is greater than 1
 - *Doesn't ensure mutual exclusion!!!*
- Example: a bounded queue for communicating processes
 - (From the well known producer-consumer problem)
- One semaphore to represent how much data is in the queue
 - Used by readers; passively blocks readers when no data available
 - Writers signal every time more data is added
- One semaphore to represent how much space is in the queue
 - Used by writers; passively blocks writers when no space available
 - Readers signal every time data is removed

Binary Semaphores

- Bounded-buffer example has two semaphores
 - One for readers, one for writers
 - Each semaphore uses critical sections for internal updates...
 - Collection of semaphores and bounded-buffer contents must also be manipulated atomically
- Use a third semaphore to enforce mutual exclusion
 - At most one process may hold this semaphore at a time
 - Processes call `wait()` before entering the critical section
 - Processes call `signal()` when leaving the critical section
- Called **binary semaphores** when used this way
 - Maximum value of semaphore is 1
 - Used to enforce mutual exclusion

Mutexes

- **Mutexes** are simplified versions of binary semaphores
 - Short for “mutual exclusion lock”
- Main difference between mutexes and binary semaphores is concept of a process “owning” a mutex when it’s locked
 - e.g. one process can’t lock a mutex and another process unlocks it
- As with semaphores, mutexes are frequently formulated to block processes until the mutex is acquired
 - Processes passively wait for mutex; can’t be used in interrupt ctxt
 - Spin locks are mutexes that actively wait
- Usually implemented with atomic test-and-set instructions
 - e.g. on IA32, **xchg** or **bts** instructions can be used to create a very efficient mutex

Other Synchronization Details

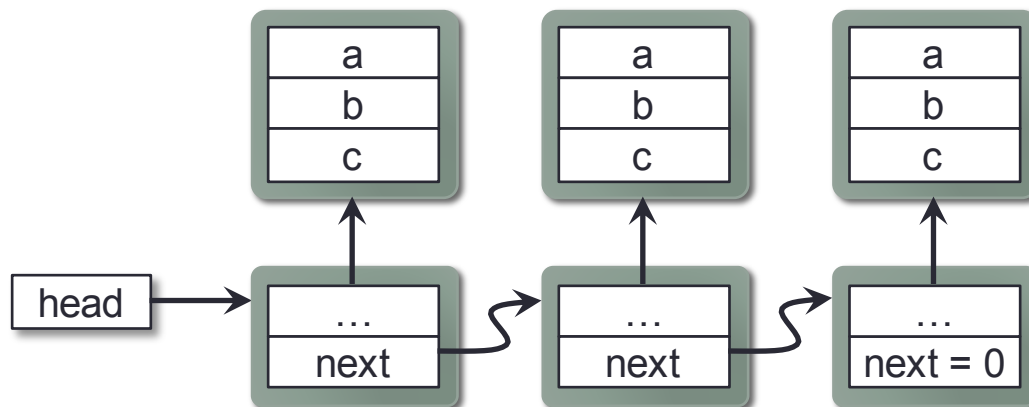
- Have only scratched the surface of synchronization
- Other thread synchronization primitives:
 - Condition variables, monitors, barriers, ...
- Classic synchronization problems:
 - The producer-consumer problem (aka the bounded buffer problem)
 - The readers-writers problem (read-write locks)
 - Dining philosophers
- Much more detail on deadlock detection and resolution
- Other multithreading difficulties:
 - Livelock, starvation, fairness, ...
- Unfortunately, beyond scope of the class to cover it all 😞

Thinking Like a Kernel Programmer

- Kernel programming is pretty different from application programming
 - Not just because it's closer to the hardware, and harder to debug...
- Very strong focus on efficiency
 - Want to minimize both memory and computing overheads
 - Ideally without sacrificing ease of maintenance
- Typically, system code simply has fewer resources to use!
 - Often, no general-purpose heap allocator in the kernel
 - Not a very large memory pool to utilize, anyway
- Also, performance issues in the kernel affect *everybody*
 - Interrupts need to complete as fast as possible
 - Process/thread scheduler needs to run quickly

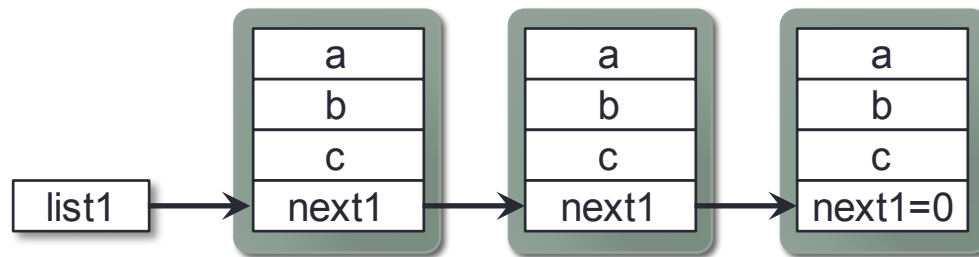
Data Structures

- Example problem: manage a linked list of items
- Typical application-programming approach:
 - Each linked-list node is separately allocated; nodes linked together
 - Often, list holds pointers to other separately-allocated structures
- In the kernel, there is often no general-purpose allocator
 - Kernels allocate a very specific and limited set of data structures
 - Can't afford to lose space required to manage heap structures, etc.



Data Structures (2)

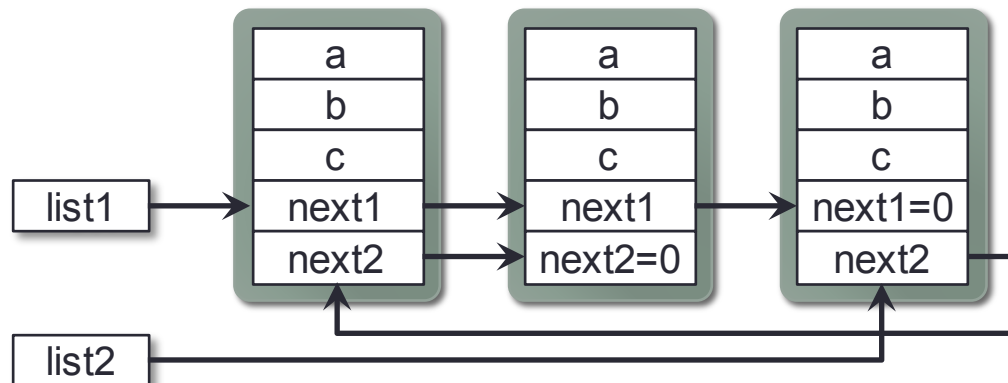
- In the kernel, collection support is often folded into the data structures being managed



- List pointers point to the “next1” member of the structure, not the start of the structure in the linked list
 - Necessary for several reasons, e.g. many different data structures might be organized into linked lists
- Requires a simple computation to get to the start of the structure from the next1 pointer
 - e.g. `(struct_t *) ((uint8_t *) ptr - offsetof(struct_t, next1))`

Data Structures (3)

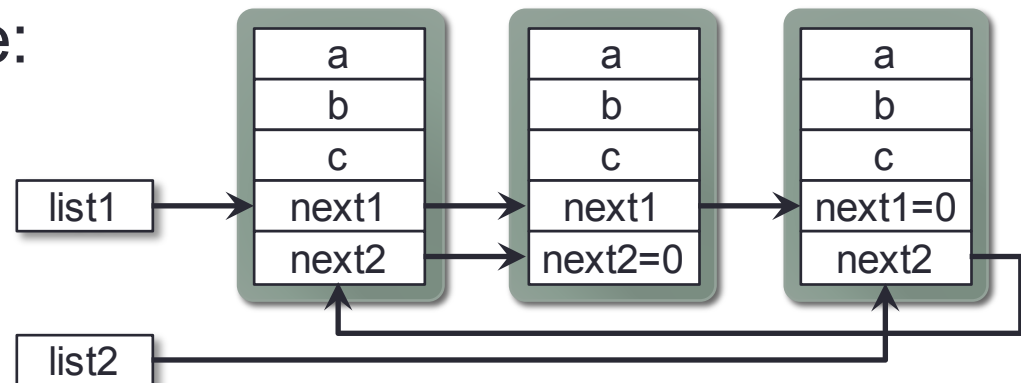
- Lists also point to the “next” pointer rather than the start of the structure, to allow for multiple lists



- Fortunately, this can be wrapped in helpful macros to simplify type declarations, list traversal, etc.

Memory Allocations

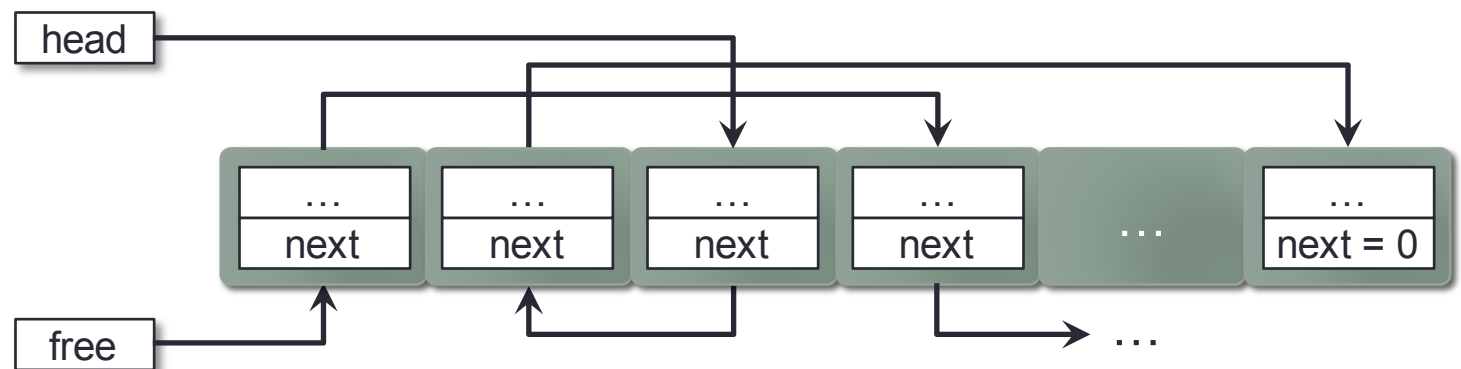
- Is it actually necessary to dynamically allocate memory?
 - Pains are taken to avoid having to do so!
- Our previous example:



- Corresponds to Pintos thread queues
 - Anchors of lists are statically-allocated global variables
 - List elements are **thread** structs, which are positioned at the lowest address of the kernel thread's stack space
- No dynamic memory allocation is required at all

Memory Allocations (2)

- Another example: need to manage a linked list of nodes
 - Linked list has a maximum size
 - Most of the time, linked list will be full or mostly full
- Instead of dynamically allocating each list node, statically allocate an array of nodes
 - Array size is maximum linked list size
- Avoids heap-management overhead (both space + time)
- May also require a list of nodes available for use



Interrupt Handlers

- Interrupt handlers need to run as quickly as possible
 - Maintains overall system responsiveness
- Often have a choice between doing work in interrupt context, vs. in process context
- Generally want to make one process wait, instead of making the entire system wait
- Pintos project 3 “thread sleep” functionality has good examples of this approach

Interrupt Handlers (2)

- Example: state for allowing threads to sleep
- Threads can store the “clock tick” when they went to sleep, plus the amount of time to sleep
 - From these values, interrupt handler can compute whether it's time for a thread to wake up or not
- Or, threads can simply store the “clock tick” when they should wake up
 - Interrupt handler doesn't have to compute anything – it just looks to see if it's time to wake up a given thread
- Second approach does computation in process context, not interrupt context
 - Only slows down the process that wants to sleep, not all timer interrupts
- Also requires less space in thread structs, which is good!

Interrupt Handlers (3)

- Example: storing sleeping threads
- Sleeping threads are all stored in one list
 - The queue of threads blocked on the timer
- Can store threads in no particular order
- Or, store threads in increasing order of wake-up time
- First approach is fast for the sleeping process
 - Just stick the process' kernel-thread onto back of the sleep-queue
 - Interrupt handler must examine all threads in the sleep-queue
- Second approach is fast for the interrupt handler
 - Sleeping process must insert kernel-thread into the proper position within the sleep queue
 - Interrupt handler knows when it can stop examining threads – when it reaches the first thread that doesn't need to wake up

Interrupt Handlers (4)

- Example: storing sleeping threads
- Generally, want to choose the second approach
 - Only the sleeping process is delayed by inserting in the proper location
 - The timer interrupt can run as fast as possible
- Aside: for priority-scheduling approach, not such a great idea to order threads based on priority
 - Thread priorities can change at any time, based on other threads' lock/unlock operations
 - Would become prohibitively expensive to maintain threads in priority order all the time

Exploiting System Constraints

- Finally, kernel code frequently exploits system constraints to use as little memory as possible
- Example: for scheduling purposes, threads are only ever in one queue
 - Which queue depends on their state
 - Either in the “ready” queue, or in some “blocked” queue, depending on what the thread is blocked on
- In these cases, can simply reuse fields for these various mutually-exclusive scenarios
 - e.g. only have one linked-list field for representing the thread’s “current queue”

Summary: Kernel Programming

- Some of these approaches yield big savings... but many of them yield small savings
 - e.g. performing computations in process-context vs. interrupt handler
- Operating systems are large, complex pieces of software
 - These small savings throughout the OS accumulate in a big way!
- Key kernel-programming question: *“How can I do things more efficiently?”*
 - Can I avoid dynamic memory allocation?
 - Can I move computations out of interrupt handlers and into process context?
 - Are there constraints on system behavior that I can exploit?
 - Can I reuse fields or data structures for multiple purposes?
- Often there are elegant solutions that also result in significantly improved system performance

Next Time

- A novel approach to the synchronization problem...